

[Table of Contents](#)

[BackCover](#)

[Data Structures Demystified](#)

[Introduction](#)

[Chapter 1: Memory, Abstract Data Types, and Addresses](#)

[Data and Memory](#)

[Reserving Memory](#)

[Memory Addresses](#)

[Quiz](#)

[Chapter 2: The Point About Variables and Pointers](#)

[Pointers](#)

[Quiz](#)

[Chapter 3: What Is an Array?](#)

[Declaring an Array](#)

[Multidimensional Arrays](#)

[Pointers and Arrays](#)

[An Array of Pointers](#)

[An Array of Pointers to Pointers](#)

[Quiz](#)

[Chapter 4: Stacks Using an Array](#)

[Inside a Stack](#)

[Creating a Stack in C++](#)

[Creating a Stack in Java](#)

[Stack in Action Using C++](#)

[Stack in Action Using Java](#)

[Quiz](#)

[Chapter 5: Queues Using an Array](#)

[Queues Using an Array in C++](#)

[Queues Using An Array in Java](#)

[Quiz](#)

[Chapter 6: What Is a Linked List?](#)

[The Structure of a Linked List](#)

[Linked Lists Using C++](#)

[Linked Lists Using Java](#)

[Quiz](#)

[Chapter 7: Stacks Using Linked Lists](#)

[LinkedList Class](#)

[The StackLinkedList Class](#)

[StackLinked List Using C++](#)

[StackLinked List Using Java](#)

[Quiz](#)

[Chapter 8: Queues Using Linked Lists](#)

[The Linked List Queue](#)

[Linked List Queue Using C++](#)

[Linked List Queue Using Java](#)

[Quiz](#)

[Chapter 9: Stacks and Queues: Insert, Delete, Peek, Find](#)

[Enhanced LinkedList Class Using C++](#)

[Enhanced LinkedList Class Using Java](#)

[Quiz](#)

[Chapter 10: What Is a Tree?](#)

[Parts of a Binary Tree](#)

[Why Use a Binary Tree?](#)

[Creating a Binary Tree](#)

[Binary Tree Using C++](#)

[Binary Tree Using Java](#)

[Quiz](#)

[Chapter 11: What Is a Hashtable?](#)

[Developing a Hashtable](#)

[Hashtable Using C++](#)

[Hashtable Using Java](#)

[Quiz](#)

[Final Exam](#)

[Index](#)

[Index_A](#)

[Index_B](#)

[Index_C](#)

[Index_D](#)

[Index_E](#)

[Index_F](#)

[Index_G](#)

[Index_H](#)

[Index_I](#)

[Index_J](#)

[Index_K](#)

[Index_L](#)

[Index_M](#)

[Index_N](#)

[Index_O](#)

[Index_P](#)

[Index_Q](#)

[Index_R](#)

[Index_S](#)

[Index_T](#)

[Index_U](#)

[Index_V](#)

[Index_W](#)

[List of Figures](#)

[List of Tables](#)

Data Structures Demystified

by Jim Keogh and Ken Davidson

ISBN:0072253592

McGraw-Hill/Osborne © 2004



Whether you are an entry-level or seasoned designer or programmer, learn all about data structures in this easy-to-understand, self-teaching guide that can be directly applied to any programming language.

Table of Contents

[Data Structures Demystified](#)

[Introduction](#)

[Chapter 1](#) - Memory, Abstract Data Types, and Addresses

[Chapter 2](#) - The Point About Variables and Pointers

[Chapter 3](#) - What Is an Array?

[Chapter 4](#) - Stacks Using an Array

[Chapter 5](#) - Queues Using an Array

[Chapter 6](#) - What Is a Linked List?

[Chapter 7](#) - Stacks Using Linked Lists

[Chapter 8](#) - Queues Using Linked Lists

[Chapter 9](#) - Stacks and Queues: Insert, Delete, Peek, Find

[Chapter 10](#) - What Is a Tree?

[Chapter 11](#) - What Is a Hashtable?

[Final Exam](#)

[Index](#)

[List of Figures](#)

[List of Tables](#)

Back Cover

If you've been searching for that quick, easy-to-understand guide to walk you through data structures, look no further. *Data Structures Demystified* is all these things and more. Whether you're trying to program stacks and linked lists or figure out hash tables, here you'll find step-by-step instructions to get the job done fast.

No longer will you have to wade through thick, dry, academic tomes, heavy on technical language and information you don't need. In *Data Structures Demystified*, each chapter starts off with an example from everyday life to demonstrate upcoming concepts, making this a totally accessible read. The authors go a step further and offer examples at the end of the chapter illustrating what you've just learned in Java and C++.

This one-of-a-kind self-teaching text offers:

- An easy way to understand data structures
- A quiz at the end of each chapter
- A final exam at the end of the book
- No unnecessary technical jargon
- A time-saving approach

About the Authors

Jim Keogh is a member of the faculty of Columbia University, where he teaches courses on Java Application Development, and is a member of the Java Community Process Program. He developed the first e-commerce track at Columbia and became its first chairperson. Jim spent more than a decade developing advanced systems for major Wall Street firms and is also the author of several best-selling computer books.

Ken Davidson is a member of the faculty of Columbia University, where he teaches courses on Java Application Development. Ken

has spent more than a decade developing advanced systems for major international firms.

Data Structures Demystified

**James Keogh
& Ken Davidson**

McGraw-Hill/Osborne

New York Chicago San Francisco Lisbon London
Madrid Mexico City Milan New Delhi San Juan
Seoul Singapore Sydney Toronto

McGraw-Hill/Osborne
2100 Powell Street, 10th Floor
Emeryville, California 94608
U.S.A.

To arrange bulk purchase discounts for sales promotions, premiums, or fund-raisers, please contact **McGraw-Hill/Osborne** at the above address. For information on translations or book distributors outside the U.S.A., please see the International Contact Information page immediately following the index of this book.

Data Structures Demystified

Copyright © 2004 by The McGraw-Hill Companies. All rights reserved. Printed in the United States of America. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

1234567890 FGR FGR 01987654
ISBN 0-07-225359-2

Publisher
Brandon A. Nordin

Vice President & Associate Publisher
Scott Rogers

Editorial Director
Wendy Rinaldi

Project Editor
Jennifer Malnick

Acquisitions Coordinator

Athena Honore

Technical Editor

Jeff Kent

Copy Editor

Sally Engelfried

Proofreader

Linda Medoff

Indexer

Claire Splan

Composition

Jean Butterfield, Tara A. Davis

Illustrators

Kathleen Edwards, Melinda Lytle

Cover Series Design

Margaret Webster-Shapiro

Cover Illustration

Lance Lekander

This book was composed with Corel VENTURA™ Publisher.

Information has been obtained by **McGraw-Hill**/Osborne from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, **McGraw-Hill**/Osborne, or others, **McGraw-Hill**/Osborne does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from the use of such information.

This book is dedicated to Anne, Sandy, Joanne, Amber-Leigh Christine, and Graaf, without whose help and support this book couldn't be written.

—Jim

To Janice, Jack, Alex, and Liz.

—Ken

About the Authors

Jim Keogh is a member of the faculty of Columbia University, where he teaches courses on Java Application Development, and is a member of the Java Community Process Program. He developed the first e-commerce track at Columbia and became its first chairperson. Jim

spent more than a decade developing advanced systems for major Wall Street firms and is also the author of several best-selling computer books.

Ken Davidson is a member of the faculty of Columbia University, where he teaches courses on Java Application Development. Ken has spent more than a decade developing advanced systems for major international firms.

Introduction

This book is for everyone who wants to learn basic data structures using C++ and Java without taking a formal course. It also serves as a supplemental classroom text. For the best results, start at the beginning and go straight through.

If you are confident about your basic knowledge of how computer memory is allocated and addressed, then skip the first two chapters, but take the quiz at the end of those chapters to see if you are actually ready to jump into data structures.

If you get 90 percent of the answers correct, you're ready. If you get 75 to 89 percent correct, skim through the text of [Chapters 1](#) and [2](#). If you get less than 75 percent of the answers correct, then find a quiet place and begin reading [Chapters 1](#) and [2](#). Doing so will get you in shape to tackle the rest of the chapters on data structures. In order to learn data structures, you must have some computer programming skills—computer programming is the language used to create data structures. But don't be intimidated; none of the programming knowledge you need goes beyond basic programming in C++ and Java.

This book contains a lot of practice quizzes and exam questions, which are similar to the kind of questions used in a data structures course. You may and should refer to the chapter texts when taking them. When you think you're ready, take the quiz, write down your answers, and then give your list of answers to a friend. Have your friend tell you your score, but not which questions were wrong. Stay with one chapter until you pass the quiz. You'll find the answers in Appendix B.

There is a final exam in Appendix A, at the end of the book, with practical questions drawn from all chapters of this book. Take the exam when you have finished all the chapters and have completed all the quizzes. A satisfactory score is at least 75 percent correct answers. Have a friend tell you your score without letting you know which questions you missed on the exam.

We recommend that you spend an hour or two each day; expect to complete one chapter each week. Don't rush. Take it at a steady pace. Take time to absorb the material. You'll complete the course in a few months; then you can use this book as a comprehensive permanent reference.

Chapter 1: Memory, Abstract Data Types, and Addresses

What is the maximum number of tries you'd need to find your name in a list of a million names? A million? No, not even close. The answer is 20—if you structure the list to make it easy to search and if you search the structure with an efficient searching technique. Searching lists is one of the many ways data structures help you manipulate data that is stored in your computer's memory. However, before you can understand how to use data structures, you need to have a firm grip on how computer memory works. In this chapter, you'll explore what computer memory is and why only zeros and ones are stored in memory. You'll also learn what a Java data type is and how to select the best Java data type to reserve memory for data used by your program.

A Tour of Memory

Computer memory is divided into three sections: main memory, cache memory in the central processing unit (CPU), and persistent storage. *Main memory*, also called *random access memory* (RAM), is where instructions (programs) and data are stored. Main memory is volatile; that is, instructions and data contained in main memory are lost once the computer is powered down.

Cache memory in the CPU is used to store frequently used instructions and data that either is, will be, or has been used by the CPU. A segment of the CPU's cache memory is called a register. A *register* is a small amount of memory within the CPU that is used to temporarily store instructions and data.

A bus connects the CPU and main memory. A *bus* is a set of etched wires on the motherboard that is similar to a highway and transports instructions and data between the CPU, main memory, and other devices connected to a computer (see [Figure 1-1](#)).

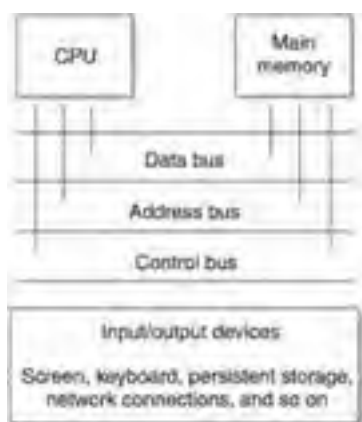


Figure 1-1: A bus connects the CPU, main memory, persistent storage, and other devices.

Persistent storage is an external storage device such as a hard disk that stores instructions and data. Persistent storage is nonvolatile; that is, instructions and data remain stored even when the computer is powered down.

Persistent storage is commonly used by the operating system as virtual memory. *Virtual memory* is a technique an operating system uses to increase the main memory capacity beyond the random access memory (RAM) inside the computer. When main memory capacity is exceeded, the operating system temporarily copies the contents of a block of memory to persistent storage. If a program needs access to instructions or data contained in the block, the operating system switches the block stored in persistent storage with a block of main memory that isn't being used.

CPU cache memory is the type of memory that has the fastest access speed. A close second is main memory. Persistent storage is a distant third because persistent storage devices usually involve a mechanical process that inhibits the quick transfer of instructions

and data.

Throughout this book, we'll focus on main memory because this is the type of memory used by data structures (although the data structures and techniques presented can also be applied to file systems on persistent storage).

Data and Memory

Data used by your program is stored in memory and manipulated by various data structure techniques, depending on the nature of your program. Let's take a close look at main memory and how data is stored in memory before exploring how to manipulate data using data structures.

Memory is a bunch of electronic switches called *transistors* that can be placed in one of two states: on or off. The state of a switch is meaningless unless you assign a value to each state, which you do using the binary numbering system.

The *binary numbering system* consists of two digits called *binary digits* (bits): zero and one. A switch in the off state represents zero, and a switch in the on state represents one. This means that one transistor can represent one of two digits.

However, two digits don't provide you with sufficient data to do anything but store the number zero or one in memory. You can store more data in memory by logically grouping together switches. For example, two switches enable you to store two binary digits, which gives you four combinations, as shown [Table 1-1](#), and these combinations can store numbers 0 through 3. Digits are zero-based, meaning that the first digit in the binary numbering system is zero, not 1. Memory is organized into groups of eight bits called a *byte*, enabling 256 combinations of zeros and ones that can store numbers from 0 through 255.

Table 1-1: Combinations of Two Bits and Their Decimal Value Equivalents

Switch 1	Switch 2	Decimal Value
0	0	0
0	1	1
1	0	2
1	1	3

The Binary Numbering System

A *numbering system* is a way to count things and perform arithmetic. For example, humans use the decimal numbering system, and computers use the binary numbering system. Both these numbering systems do exactly the same thing: they enable us to count things and perform arithmetic. You can add, subtract, multiply, and divide using the binary numbering system and you'll arrive at the same answer as if you used the decimal numbering system.

However, there is a noticeable difference between the decimal and binary numbering systems: the decimal numbering system consists of 10 digits (0 through 9) and the binary

numbering system consists of 2 digits (0 and 1).

To jog your memory a bit, remember back in elementary school when the teacher showed you how to “carry over” a value from the right column to the left column when adding two numbers? If you had 9 in the right column and added 1, you changed the 9 to a 0 and placed a 1 to the left of the 0 to give you 10:

$$\begin{array}{r} 9 \\ +1 \\ \hline 10 \end{array}$$

The same “carry over” technique is used when adding numbers in the binary numbering system except you carry over when the value in the right column is 1 instead of 9. If you have 1 in the right column and add 1, you change the 1 to a 0 and place a 1 to the left of the 0 to give you 10:

$$\begin{array}{r} 1 \\ +1 \\ \hline 10 \end{array}$$

Now the confusion begins. Both the decimal number and the binary number seem to have the same value, which is ten. Don't believe everything you see. The decimal number does represent the number 10. However, the binary number 10 isn't the value 10 but the value 2.

The digits in the binary numbering system represent the state of a switch. A computer performs arithmetic by using the binary numbering system to change the state of sets of switches.

Reserving Memory

Although a unit of memory holds a byte, data used in a program can be larger than a byte and require 2, 4, or 8 bytes to be stored in memory. Before any data can be stored in memory, you must tell the computer how much space to reserve for data by using an abstract data type.

An *abstract data type* is a keyword of a programming language that specifies the amount of memory needed to store data and the kind of data that will be stored in that memory location. However, an abstract data type does not tell the computer how many bytes to reserve for the data. The number of bytes reserved for an abstract data type varies, depending on the programming language used to write the program and the type of computer used to compile the program.

Abstract data types in Java have a fixed size in order for programs to run in all Java runtime environments. In C and C++, the size of an abstract data type is based on the register size of the computer used to compile the program. The `int` and `float` data types are the size of the register. A `short` data type is half the size of an `int`, and a `long` data type is double the size of an `int`.

Think of an abstract data type as the term “case of tomatoes.” You call the warehouse manager and say that you need to reserve enough shelf space to hold five cases of tomatoes. The warehouse manager knows how many shelves to reserve because she knows the size of a case of tomatoes.

The same is true about an abstract data type. You tell the computer to reserve space for an integer by using the abstract data type `int`. The computer already knows how much memory to reserve to store an integer.

The abstract data type also tells the computer the kind of data that will be stored at the memory location. This is important because computers manipulate data of some abstract data types differently than data of other abstract data types. This is similar to how the warehouse manager treats a case of paper plates differently than a case of tomatoes.

[Table 1-2](#) contains a list of abstract data types. The first column contains keywords for each abstract data type. The second column lists the corresponding number of bits that are reserved in memory for a Java program. The third column shows the range of values that can be stored in the abstract data type. And the last column is the group within which the abstract data type belongs.

Table 1-2: Simple Java Data Types.

Data Type	Data Type Size in Bits	Range of Values	Group

byte 8	8	-128 to 127	Integers
short 16	16	-32,768 to 32,767	Integers
int 32	32	-2,147,483,648 to 2,147,483,647	Integers
long 64	64	- 9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Integers
char 16 (Unicode)	16 (Unicode)	65,536 (Unicode)	Characters
float 32	32	3.4e-038 to 3.4e+038	Floating-point
double 64	64	1.7e-308 to 1.7e+308	Floating-point
boolean 1	1	0 or 1	Boolean

You choose the abstract data type that best suits the data that you want stored in memory, then use the abstract data type in a declaration statement to declare a variable. A variable is a reference to the memory location that you reserved using the declaration statement (see [Chapter 2](#)).

You should always reserve the proper amount of memory needed to store data because you might lose data if you reserve too small a space. This is like sending ten cases of tomatoes to the warehouse when you only reserved space for five cases. If you do this, the other five cases will get tossed aside.

Abstract Data Type Groups

You determine the amount of memory to reserve by determining the appropriate abstract data type group to use and then deciding which abstract data type within the group is right for the data.

There are four data type groups:

- **Integer** Stores whole numbers and signed numbers. Great for storing the number of dollars in your wallet when you don't need a decimal value.
- **Floating-point** Stores real numbers (fractional values). Perfect for storing bank deposits where pennies (fractions of a dollar) can quickly add up to a few dollars.

- **Character** Stores a character. Ideal for storing names of things.
- **Boolean** Stores a `true` or `false` value. The correct choice for storing a yes or no or true or false response to a question.

Integers

The *integer* abstract data type group consists of four abstract data types used to reserve memory to store whole numbers: `byte`, `short`, `int`, and `long`, as described in [Table 1-2](#).

Depending on the nature of the data, sometimes an integer must be stored using a positive or negative sign, such as a `+10` or `-5`. Other times an integer is assumed to be positive so there isn't any need to use a positive sign. An integer that is stored with a sign is called a *signed number*; an integer that isn't stored with a sign is called an *unsigned number*.

What's all this hoopla about signed numbers? The sign takes up 1 bit of memory that could otherwise be used to represent a value. For example, a `byte` has 8 bits, all of which can be used to store an unsigned number from 0 to 255. You can store a signed number in the range of `-128` to `+127`.

C and C++ support unsigned integers. Java does not. An *unsigned integer* is a value that is implied to be positive. The positive sign is not stored in memory. All integers in Java are represented with a sign. Zero is stored as a positive number.

byte Abstract Data Type

The `byte` abstract data type is the smallest abstract data type in the integer group and is declared by using the keyword `byte` (see [Figure 1-2](#)). Programmers typically use a `byte` abstract data type when sending data to and receiving data from a file or across a network. The `byte` abstract data type is also commonly used when working with binary data that may not be compatible with other abstract data types. Choose a `byte` whenever you need to move data to and from a file or across a network.

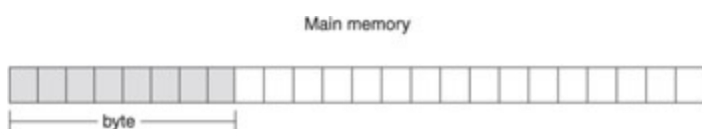


Figure 1-2: A `byte` abstract data type in Java reserves 8 bits of main memory.

short Abstract Data Type

The `short` abstract data type is ideal for use in programs that run on 16-bit computers. However, most of those computers are on the trash heap and have been replaced by 32-bit and 64-bit computers! (See [Figure 1-3](#).) Therefore, the `short` is the least used integer

abstract data type. Choose a `short` if you ever need to store an integer in a program that runs on a very old computer.

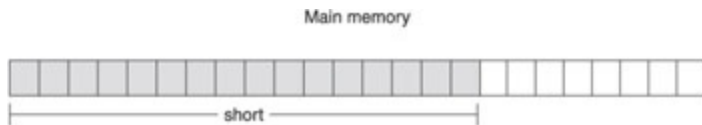


Figure 1-3: A `short` abstract data type in Java reserves 16 bits of main memory.

int Abstract Data Type

The `int` abstract data type is the most frequently used abstract data type of the integer group for a number of reasons (see [Figure 1-4](#)). Choose an `int` :

- For control variables in control loops
- In array indexes
- When performing integer math

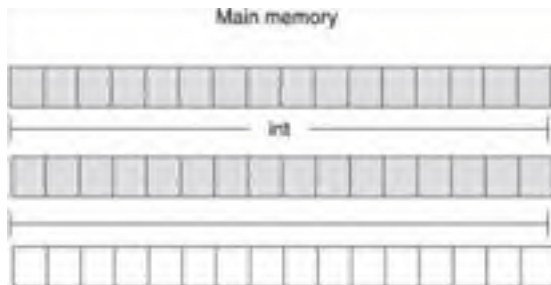


Figure 1-4: An `int` abstract data type in Java reserves 32 bits of main memory.

long Abstract Data Type

A `long` abstract data type (see [Figure 1-5](#)) is used whenever using whole numbers that are beyond the range of an `int` data type (refer to [Table 1-2](#)). Choose a `long` when storing the net worths of Bill Gates, Warren Buffet, and you in a program.

Floating-Point

Abstract data types in the *floating-point* group are used to store real numbers in memory. A *real number* contains a decimal value. There are two kinds of floating point data types: `float` and `double` (as described in [Table 1-2](#)). The `float` abstract data type is a single precision number, and a `double` is a double precision number. *Precision* of a number is the number of places after the decimal point that contains an accurate value.

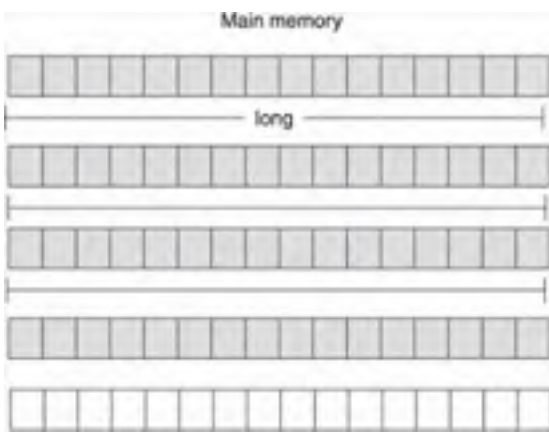


Figure 1-5: A `long` abstract data type in Java reserves 64 bits of main memory.

The term *floating-point* refers to the way decimals are referenced in memory. There are two parts of a floating-point number: the real number, which is stored as a whole number, and the position of the decimal point within the whole number. This is why it is said that the decimal point “floats” within the number.

For example, the floating-point value 43.23 is stored as 4323 (no decimal point). Reference is made in the number indicating that the decimal point is placed after the second digit.

float Abstract Data Type

The `float` abstract data type (see [Figure 1-6](#)) is used for real numbers that require single precision, such as United States currency. *Single precision* means the value is precise up to 7 digits to the right of the decimal. For example, suppose you divide \$53.50 evenly among 17 people. Each person would get \$3.147058823529. Digits to the right of \$3.1470588 are not guaranteed to be precise because of the way a `float` is stored in memory. Choose a `float` whenever you need to store a decimal value where only 7 digits to the right of the decimal must be accurate.

double Abstract Data Type The `double` abstract data type (see [Figure 1-7](#)) is used to store real numbers that are very large or very small and require double the amount of memory that is reserved with a `float` abstract data type. Choose a `double` whenever you need to store a decimal value where more than 7 digits to the right of the decimal must be accurate.

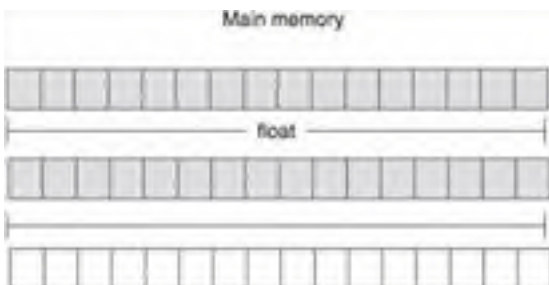


Figure 1-6: A `float` abstract data type in Java reserves 32 bits of main memory.

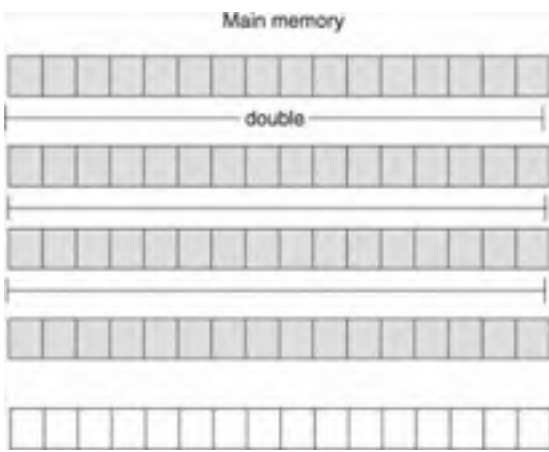


Figure 1-7: A `double` abstract data type in Java reserves 64 bits of main memory.

Characters

A *character* abstract data type (see [Figure 1-8](#)) is represented as an integer value that corresponds to a character set. A *character set* assigns an integer value to each character, punctuation, and symbol used in a language.



Figure 1-8: A `char` abstract data type in Java reserves 16 bits of main memory.

For example, the letter *A* is stored in memory as the value 65, which corresponds to the letter *A* in a character set. The computer knows to treat the value 65 as the letter *A* rather than the number 65 because memory was reserved using the `char` abstract data type. The keyword `char` tells the computer that the integer stored in that memory location is treated as a character and not a number.

There are two character sets used in programming, the American Standard Code for Information Interchange (ASCII) and Unicode. ASCII is the granddaddy of character sets and uses a byte to represent a maximum of 256 characters. However, a serious problem was evident after years of using ASCII. Many languages such as Russian, Arabic, Japanese, and Chinese have more than 256 characters in their language. A new character set called Unicode was developed to resolve this problem. Unicode uses 2 bytes to represent each character. Choose a `char` whenever you need to store a single character in memory.

Boolean Abstract Data Type

A `boolean` abstract data type (see [Figure 1-9](#)) reserves memory to store a `boolean` value, which is a `true` or `false` represented as a zero or one. Choose a `boolean` whenever you need to store one of two possibilities in memory.

Main memory

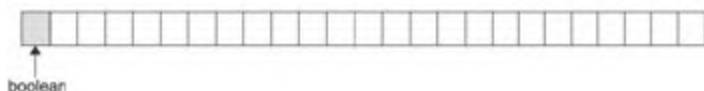


Figure 1-9: A `boolean` abstract data type in Java reserves 1 bit of main memory.

Memory Addresses

Imagine main memory as a series of seemingly endless boxes organized into groups of eight. Each box holds a zero or one. Each group of eight boxes (1 byte) is assigned a unique number called a *memory address*, as shown in [Figure 1-10](#). It is very important to keep this in mind as you learn about data structures; otherwise, you can easily become confused.

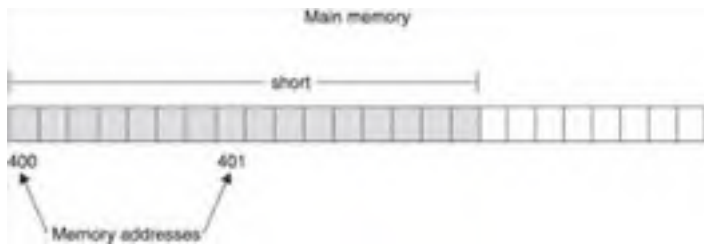


Figure 1-10: The memory address of the first byte is used to reference all bytes reserved for an abstract data type.

A memory address is indirectly or directly used within a program to access all eight boxes. For example, say your program tells the computer that you want to copy data stored in memory location 423—that is, the box whose address is 423. The computer goes to that memory location and copies the data (zero or one) from box 423 and copies data from the next seven boxes. Those next seven boxes don't have a memory address. You could say that those seven boxes share the memory address of box 423.

Real Memory Addresses

Memory addresses are represented so far throughout this chapter as a decimal value, such as “box 423.” In reality, memory addresses are a 32-bit or 64-bit number, depending on the computer's operating system, and are represented as a hexadecimal value.

Hexadecimal is a numbering system similar to the decimal and binary numbering systems. That is, hexadecimal values are used to count and they are used in arithmetic. The hexadecimal numbering system has 16 digits from 0 through 9 and A through F, which represents 10 through 15. Here is how memory address 258,425,506 is represented in hexadecimal notation 0x0F6742A2.

Abstract Data Types and Memory Addresses











Previously in this chapter you learned that you reserve memory for data by using an abstract data type. Some abstract data types reserve memory in a size that is greater than 1 byte. For example, the `short` abstract data type in Java reserves 2 bytes of memory.

Since each byte of memory has its own memory address, you might assume a `short` has two memory addresses because it uses 2 bytes of memory. That's not the case. The

computer uses the memory address of the first byte to reference any abstract data type that reserves multiple bytes of memory.

Let's say that space was reserved in memory for a `short` abstract data type (see [Figure 1-10](#)). Two memory locations are reserved, memory addresses 400 and 401. However, only memory address 400 is used to reference the `short`. The computer automatically knows that the value stored in memory address 401 is part of the value stored in memory address 400 because the space was reserved using an `short` abstract data type. Therefore, the computer copies all the bits from memory address 400 *and* all the bits from memory address 401 whenever a request is made by the program to copy the integer stored at memory address 400.

Quiz

1. What is an abstract data type? 
2. What abstract data type would be used to store a whole number? 
3. Explain how a memory address is used to access an abstract data type that is larger than 1 byte. 
4. What is the difference between a `float` abstract data type and a `double` abstract data type? 
5. What is precision? 
6. Explain how memory is organized within a computer. 
7. What is a numbering system? 
8. Why is the binary numbering system used in computing? 
9. Why don't you directly specify the number of bytes to reserve in memory to store data? 
10. Explain the impact signed and unsigned numbers have on memory. 

Answers

1. An abstract data type is a keyword of a programming language that specifies the amount of memory needed to store data and the kind of data that will be stored in that memory location.
2. An integer: `byte`, `short`, `int`, or `long`.

Each memory address represents 1 byte of memory. Some abstract data types, such as an `int`, reserve 2 bytes of memory. Technically, data stored in this memory location has two memory address: one address for the first byte of memory and another address for the second byte of memory. However, the computer references only the address of the first byte of memory when accessing that memory location.
- 3.
4. The `double` abstract data type is used to store real numbers that are very large or very small and require double the amount of memory that is reserved with a `float` abstract data type.
5. Precision refers the accuracy of the decimal portion of a value.

Memory consists of a series of switches called transistors. Each transistor stores a binary digit (bit). Transistors are logically organized into groups of 8 switches called a byte. Each byte is uniquely identified by a memory address.

6.

A numbering system is a logical method used to count and perform arithmetic using digits to represent items. Each numbering system has a different number of digits. The decimal numbering system has 10 digits, from 0 through 9. The binary numbering system has 2 digits, 0 and 1. All numbering systems can be used to count and perform arithmetic, regardless of the number of digits contained in the numbering system.

7.

The binary numbering system is used in computing because it contains 2 digits that can be stored by changing the state of a transistor. Off represents 0 and On represents 1.

8.

A programmer doesn't specify the exact number of bytes to reserve in memory because the computer language determines the most efficient number of bytes to represent a data type.

9.

The sign takes up 1 bit of memory that could otherwise be used to represent a value. For example, a byte has 8 bits, all of which can be used to store an unsigned number from 0 to 255. You can store a signed number in the range of -128 to +127.

10.

Chapter 2: The Point About Variables and Pointers

Some programmers cringe at the mere mention of the word “pointer” because it brings to mind complex, low-level programming techniques that are confounding. Hogwash. Pointers are child play, literally. Watch a 15-month-old carefully and you’ll notice that she points to things she wants, and that’s a pointer in a nutshell. A pointer is a variable that is used to point to a memory address whose content you want to use in your program. You’ll learn all about pointer variables in this chapter.

Declaring Variables and Objects

Memory is reserved by using a data type in a declaration statement. The form of a declaration statement varies depending on the programming language you use. Here is a declaration statement for C, C++, and Java:

```
int myVariable;
```

There are three parts to this declaration statement:

- **Data type** Tells how much memory to reserve and the kind of data that will be stored in that memory location
- **Variable name** A name used within the program to refer to the contents of that memory location
- **Semicolon** Tells the computer this is an instruction (statement)

Primitive Data Types and User-Defined Data Types

In [Chapter 1](#), you were introduced to the concept of abstract data types, which are used to reserve computer memory. Abstract data types are divided into two categories, primitive data types and user-defined data types. A primitive data type is defined by the programming language, such as the data types you learned about in the [previous chapter](#). Some programmers call these built-in data types.

The other category of abstract data type, a user-defined data type, is a group of primitive data types defined by the programmer. For example, let's say you want to store students' grades in memory. You'll need to store 4 data elements: the student's ID, first name, last name, and grade. You could use primitive data types for each data element, but primitive data types are not grouped together; each exists as separate data elements.

A better approach is to group primitive data types into a user-defined data type to form a record. You probably heard the term "record" used when you learned about databases. Remember that a database consists of one or more tables. A table is similar to a spreadsheet consisting of columns and rows. A row is also known as a record. A user-defined data type defines columns (primitive data types) that comprise a row (a user-defined data type).

The form used to define a user-defined data type varies depending on the programming language used to write the program. Some programming languages, such as Java, do not support user-defined data types. Instead, attributes of a class are used to group together primitive data types; this is discussed later in this chapter.

In the C and C++ programming languages, you define a user-defined data type by defining a structure. Think of a structure as a stencil of the letter A. The stencil isn't the letter A, but